ECE 601: Homework 8

Wei Xu
Email: xu1639@purdue.edu
Due date: 11:59 PM, Mar. 29, 2024
(Spring 2024)

1 Programming Tasks

1.1 Generator and discriminator networks

Note: The code for this subsection is borrowed from the `DLStudio` package (https://engineering.purdue.edu/kak/distDLS/).

In this implementation, the architectures of the generator and discriminator can be found in Fig. 1, and they are defined by the class `GeneratorDCGAN` and class `DiscriminatorDCGAN` respectively. In `GeneratorDCGAN`, fake images are generated from a latent vector by upsampling with `nn.ConvTranspose2d`, while decreasing the number of channels. The first four `nn.ConvTranspose2d` layers are followed by `nn.BatchNorm2d` and `nn.ReLU`, and the last `nn.ConvTranspose2d` layer is followed by `nn.Tanh`. The tensor shape changes through the generator is $B \times 100 \times 1 \times 1 \rightarrow B \times 512 \times 4 \times 4 \rightarrow B \times 256 \times 8 \times 8 \rightarrow B \times 128 \times 16 \times 16 \rightarrow B \times 64 \times 32 \times 32 \rightarrow B \times 3 \times 64 \times 64$, where B is the batch size. In `DiscriminatorDCGAN`, labels are predicted by convolving the images with `nn.Conv2d`. The first `nn.Conv2d` layer is followed by `nn.LeakyReLU`, the following three `nn.Conv2d` layers are followed by `nn.BatchNorm2d` and `nn.LeakyReLU`, and the last `nn.Conv2d` layer is followed by `nn.Sigmoid` to predict the scores as labels. The tensor shape changes through the generator is $B \times 3 \times 64 \times 64 \rightarrow B \times 64 \times 32 \times 32 \rightarrow B \times 128 \times 16 \times 16 \rightarrow B \times 256 \times 8 \times 8 \rightarrow B \times 512 \times 4 \times 4 \rightarrow B \times 1 \times 1 \times 1$.

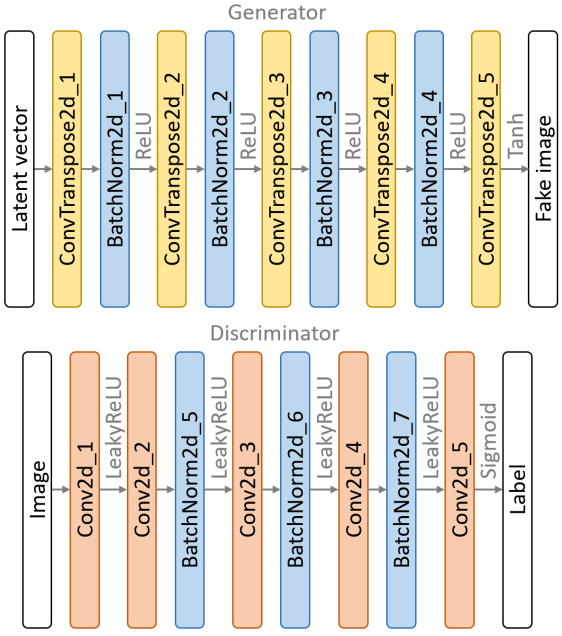

Figure 1: Architectures of `GeneratorDCGAN` and `DiscriminatorDCGAN`.

The `GeneratorDCGAN` and `DiscriminatorDCGAN` implementations are shown below. For `GeneratorDCGAN`, there are 3576704 learnable parameters and 13 layers. For `DiscriminatorDCGAN`,

there are 2766529 learnable parameters and 16 layers.

```
...
# class for generator
class GeneratorDCGAN(nn.Module):
    def __init__(self):
        super().__init__()
        # transposed convolutional layer with batch normalizaiton and ReLU activation
        def trans_conv_layer(in_channels, out_channels, kernel_size, stride=1, padding=0):
            return nn.Sequential(nn.ConvTranspose2d(in_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding, bias=False),
                                 nn.BatchNorm2d(out_channels),
                                 nn.ReLU(inplace=True))
        # define layers
        self.latent2image = trans_conv_layer(100, 512, 4, 1, 0)
        self.upsampler2 = trans_conv_layer(512, 256, 4, 2, 1)
        self.upsampler3 = trans_conv_layer(256, 128, 4, 2, 1)
        self.upsampler4 = trans_conv_layer(128, 64, 4, 2, 1)
        self.upsampler5 = nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False)
        self.tanh = nn.Tanh()
    def forward(self, z):
        z = self.latent2image(z)
        z = self.upsampler2(z)
        z = self.upsampler3(z)
        z = self.upsampler4(z)
        z = self.upsampler5(z)
        z = self.tanh(z)
        return z

# class for discriminator
class DiscriminatorDCGAN(nn.Module):
    def __init__(self):
        super().__init__()
        # convolutional layer with batch normalizaiton (optional) and ReLU activation
        def conv_layer(in_channels, out_channels, kernel_size, stride=1, padding=0, bn=True):
            if bn:
                return nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding),
                                     nn.BatchNorm2d(out_channels),
                                     nn.LeakyReLU(negative_slope=0.2, inplace=True))
            else:
                return nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding),
                                     nn.LeakyReLU(negative_slope=0.2, inplace=True))
        # define layers
```

```
        self.conv1 = conv_layer(3, 64, 4, 2, 1, False) # first
            ↪ convolutional layer does not have batch normalization
        self.conv2 = conv_layer(64, 128, 4, 2, 1)
        self.conv3 = conv_layer(128, 256, 4, 2, 1)
        self.conv4 = conv_layer(256, 512, 4, 2, 1)
        self.conv5 = nn.Conv2d(512, 1, 4, 1, 0)
        self.sig = nn.Sigmoid()
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.sig(x)
        return x
...
```

1.2 Adversarial training logic

Note: The code for this subsection is borrowed from the `DLStudio` package (https://engineering.purdue.edu/kak/distDLS/).

For the discriminator with weights θ_d, it is expected to output a higher score if the input image x is from the same distribution as the training data p_{data}. So the optimization problem for this part is

$$\max_{\theta_d} \mathbb{E}_{x \sim p_{data}}[\log D(x)] \tag{1}$$

For the generator with weights θ_g, the input latent vector z is from the probability distribution p_Z. Its goal is to generate a fake image but let the discriminator give a high score. So the optimization problem for this part is

$$\min_{\theta_g} \mathbb{E}_{z \sim p_Z}[\log(1 - D(G(z)))] \tag{2}$$

These two above can be combined as

$$\min_{\theta_g} \max_{\theta_d} [\mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{z \sim p_Z}[\log(1 - D(G(z)))]] \tag{3}$$

The function code for creating the training data loader is shown below. The images are read directly from the folder with `torchvision.datasets.ImageFolder`. `tvt.Resize` can make sure that the images have the desired size. Then normalize the image pixel values to $[-1, 1]$ with `tvt.ToTensor` and `tvt.Normalize`. Finally, return the `train_dataloader` instance for training. The batch size is 32, and the number of workers is 2.

```
...
# create data loader
def create_dataloader(args):
    dataset = torchvision.datasets.ImageFolder(root=args.imgs_path,
                    transform = tvt.Compose([tvt.Resize(args.img_size),
                                             tvt.ToTensor(),
                                             tvt.Normalize((0.5, 0.5,
                                                ↪ 0.5), (0.5, 0.5, 0.5)
                                                ↪ )
```

```
                                                ]))
    train_dataloader = DataLoader(dataset, batch_size=args.batch_size,
        ↪ shuffle=True, num_workers=args.n_workers)
    return train_dataloader
...
```

The training function code is shown below. Both generator and discriminator use `torch.optim.Adam` as the optimizers with $\beta_1 = 0.5$, $\beta_2 = 0.99$, and learning rate $lr = 1e\text{-}4$. The model weights are initialized with `weight_init` function. the number of epochs is 30. The number of channels of 1×1 input noise vectors for the generator is 100. The label for a real image is 1, and that for a fake image is 0. The loss criterion is `nn.BCELoss`. For the discriminator, since it is the maximization step, the training target for real images is 1, according to the first item in Eq. 3; while the training target for fake images is 0, according to the second item in Eq. 3. For the generator, which is only contained in the second item of Eq. 3, the minimization of that item means the maximization of $D(G(z))$, which would encourage the generator to produce high-quality fake images. Besides periodically printing the losses, qualitative tracking by saving generated fake images is also implemented, which would be used to generate a `.gif` file. The `plot_learning_curve` plots the training losses versus iteration. In addition, a batch of real and fake images are saved for the demonstration purpose.

```
...
# initialize weights
def weight_init(m):
    classname = m.__class__.__name__
    # initialize (transposed) convolutional layers
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    # initialize batch normalization layers
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)

# plot the learning curves
def plot_learning_curve(args, G_loss_list, D_loss_list):
    # generate x-axis
    x = [i for i in range(len(G_loss_list))]
    plt.figure(figsize=(10, 5))
    # plot each curve
    plt.plot(x, G_loss_list, label='Generator')
    plt.plot(x, D_loss_list, label='Discriminator')
    plt.xlabel('iterations')
    plt.ylabel('losses')
    plt.legend()
    cf = plt.gcf()
    cf.savefig('{}loss.png'.format(args.result_path), bbox_inches='
        ↪ tight', format='png')

# train the model
def train(args, device):
    # create a folder to save results
    dir_name_for_results = args.result_path
    if os.path.exists(dir_name_for_results):
        files = glob.glob(dir_name_for_results + "/*")
        for file in files:
```

```
            if os.path.isfile(file):
                os.remove(file)
            else:
                files = glob.glob(file + "/*")
                list(map(lambda x: os.remove(x), files))
    else:
        os.mkdir(dir_name_for_results)
    # create training data loader
    train_dataloader = create_dataloader(args)
    # fixed noise to track performance during training
    noise_fixed = torch.randn(args.batch_size, args.nz, 1, 1, device=
        ↪ device)
    # create model instance
    generator = GeneratorDCGAN().to(device)
    discriminator = DiscriminatorDCGAN().to(device)
    # iniitalize weights
    generator.apply(weight_init)
    discriminator.apply(weight_init)
    # count number of learnable parameters
    num_learnable_params_gen = sum(p.numel() for p in generator.
        ↪ parameters() if p.requires_grad)
    print('The number of learnable parameters in the Generator: %d'%
        ↪ num_learnable_params_gen)
    num_learnable_params_disc = sum(p.numel() for p in discriminator.
        ↪ parameters() if p.requires_grad)
    print('The number of learnable parameters in the Discriminator: %d'
        ↪ %num_learnable_params_disc)
    # count number of layers
    num_layers_gen = len(list(generator.parameters()))
    print('The number of layers in the generator: %d'%num_layers_gen)
    num_layers_disc = len(list(discriminator.parameters()))
    print('The number of layers in the discriminator: %d'%
        ↪ num_layers_disc)
    # use Adam optimizer for both networks
    optimizer_G = torch.optim.Adam(generator.parameters(), lr=args.lr,
        ↪ betas=(args.beta1,0.99))
    optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=args.
        ↪ lr, betas=(args.beta1,0.99))
    # use BCE loss
    criterion = nn.BCELoss()
    # list to save losses
    G_loss_list = []
    D_loss_list = []
    # labels for real and fake images
    real_label_val = 1
    fake_label_val = 0
    # track iteration index
    iters = 0
    # list to save generated fake images during training
    imgs_list = []
    for epoch in range(args.n_epochs):
        # list to save losses for printing
        G_loss_list_temp = []
        D_loss_list_temp = []
```

```
        for i, data in enumerate(train_dataloader):
            imgs_real = data[0].to(device)
            # actual number of images in current batch
            b_size = imgs_real.size(0)
            # create labels for real and fake images
            label_real = torch.full((b_size,), real_label_val, dtype=
                ↪ torch.float, device=device)
            label_fake = torch.full((b_size,), fake_label_val, dtype=
                ↪ torch.float, device=device)
            # train discriminator
            discriminator.zero_grad()
            output = discriminator(imgs_real).view(-1)
            D_loss_real = criterion(output, label_real)
            D_loss_real.backward() # gradient from real images
            noise = torch.randn(b_size, args.nz, 1, 1, device=device)
            fakes = generator(noise) # generator fake images
            output = discriminator(fakes.detach()).view(-1)
            D_loss_fake = criterion(output, label_fake)
            D_loss_fake.backward() # gradient from fake images
            optimizer_D.step() # update discriminator
            D_loss = D_loss_real + D_loss_fake
            D_loss_list_temp.append(D_loss)
            # train generator
            generator.zero_grad()
            output = discriminator(fakes).view(-1)
            G_loss = criterion(output, label_real)
            G_loss.backward() # gradient from generated images
            optimizer_G.step() # update generator
            G_loss_list_temp.append(G_loss)
            # periodically print losses
            if i%args.print_every == args.print_every-1:
                mean_G_loss = torch.mean(torch.FloatTensor(
                    ↪ G_loss_list_temp))
                mean_D_loss = torch.mean(torch.FloatTensor(
                    ↪ D_loss_list_temp))
                print('[epoch=%d/%d iter=%4d] mean_D_loss=%7.4f
                    ↪ mean_G_loss=%7.4f]'%((epoch+1),args.n_epochs,(i
                    ↪ +1),mean_D_loss,mean_G_loss))
                G_loss_list_temp = []
                D_loss_list_temp = []
            # save losses
            G_loss_list.append(G_loss.item())
            D_loss_list.append(D_loss.item())
            # periodically visualize generated images
            if (iters%args.generate_every == 0) or ((epoch == args.
                ↪ n_epochs-1) and (i == len(train_dataloader)-1)):
                with torch.no_grad():
                    fake = generator(noise_fixed).detach().cpu()
                imgs_list.append(torchvision.utils.make_grid(fake,
                    ↪ padding=1, pad_value=1, normalize=True))
            iters += 1
    # save the trained network
    torch.save(generator, '{}generator.pth'.format(args.result_path))
    torch.save(discriminator, '{}discriminator.pth'.format(args.
```

```
        ↪ result_path))
    # plot learning curve
    plot_learning_curve(args, G_loss_list, D_loss_list)
    # generate GIF
    imgs = []
    for imgobj in imgs_list:
        img = tvtF.to_pil_image(imgobj)
        imgs.append(img)
    imageio.mimsave('{}generation_animation.gif'.format(args.
        ↪ result_path), imgs, fps=5)
    # visualize real image samples from the dataset
    real_batch = next(iter(train_dataloader))[0]
    cv2.imwrite('{}real_imgs.png'.format(args.result_path), np.flip(
        ↪ torchvision.utils.make_grid(real_batch.to(device), padding=1,
        ↪  pad_value=1, normalize=True).cpu().numpy().transpose(1,2,0)
        ↪ ,2)*255)
    # visualize fake image samples
    cv2.imwrite('{}fake_imgs.png'.format(args.result_path), np.flip(
        ↪ imgs_list[-1].numpy().transpose(1,2,0),2).clip(0,1)*255)
...
```

The adversarial losses over training iterations for both the generator and discriminator can be found in Fig. 2.

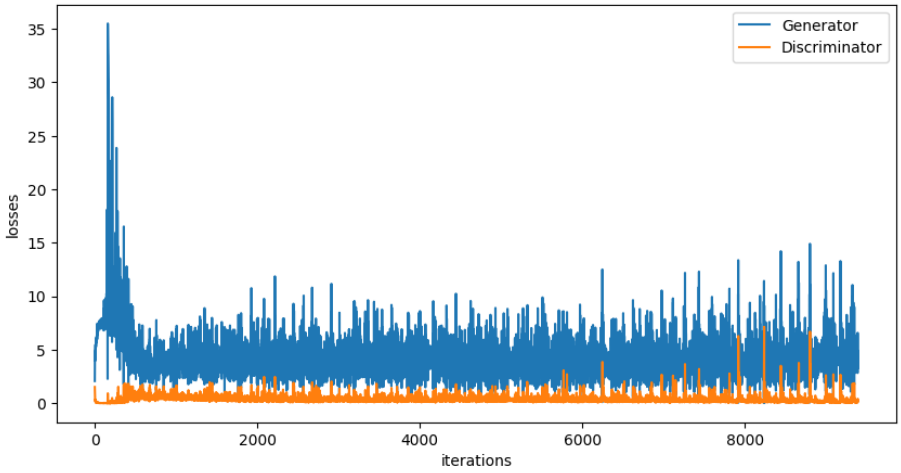

Figure 2: Adversarial losses over training iterations.

1.3 Comparison with Diffusion and evaluation

Note: The code for this subsection is borrowed from the `DLStudio` package (https://engineering.purdue.edu/kak/distDLS/).

1.3.1 Qualitative evaluation

To qualitatively evaluate the performance of the GAN and Diffusion models, plot 4×4 image grids. The code for plotting the real images and the fake images by GAN is shown below. In the `generate_img_demo` function, change the batch size `args.batch_size` to 16. And run the generator to generate 16 fake images.

```
...
# generate samples
def generate_img_demo(args, device):
    # change the batch size to 16 for the 4 by 4 demo
    args.batch_size = 16
    # create training data loader
    train_dataloader = create_dataloader(args)
    # load the trained network
    generator = torch.load('{}generator.pth'.format(args.result_path)).
        ↪ to(device).eval()
    # visualize real image samples from the dataset
    real_batch = next(iter(train_dataloader))[0]
    cv2.imwrite('{}real_imgs_demo.png'.format(args.result_path), np.
        ↪ flip(torchvision.utils.make_grid(real_batch.to(device), nrow
        ↪ =4, padding=1, pad_value=1, normalize=True).detach().cpu().
        ↪ numpy().transpose(1,2,0),2)*255)
    # visualize fake image samples
    noise = torch.randn(args.batch_size, args.nz, 1, 1, device=device)
    with torch.no_grad():
        fake = generator(noise).detach().cpu()
    cv2.imwrite('{}fake_imgs_demo.png'.format(args.result_path), np.
        ↪ flip(torchvision.utils.make_grid(fake, nrow=4, padding=1,
        ↪ pad_value=1, normalize=True).numpy().transpose(1,2,0),2).clip
        ↪ (0,1)*255)
...
```

The `VisualizeSamples.py` file in `ExamplesDiffusion` folder of `DLStudio` also needs to be modified to generate visualize 4×4 image grids and save sample with the resolution of 64×64. The modified version `VisualizeSamples_modified.py` is shown below. To be concise, only the lines that are modified or added are commented. For example, `image_display_size` is changed as $(64, 64)$; the code for saving 4×4 image grid is rewritten using `torchvision.utils.make_grid`.

```
##  VisualizeSamples_modified.py

##  Avi Kak, March 2024
##  modified by Wei Xu for HW8

import os,sys
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
import glob
from PIL import Image
import cv2 # import OpenCV package for saving image grid

print("\n\nPutting together a collage of the generated images for
    ↪ display\n")
print("\nFor the individual images, check the directory '
    ↪ visualize_samples'\n\n")

npz_archive = "RESULTS/samples_2048x64x64x3.npz"
visualization_dir  =  "visualize_samples"
image_display_size = (64,64) # changed the display size to 64 by 64

```

```
if os.path.exists(visualization_dir):
    files = glob.glob(visualization_dir + "/*")
    for file in files:
        if os.path.isfile(file):
            os.remove(file)
else:
    os.mkdir(visualization_dir)

data = np.load(npz_archive)

for i, arr in enumerate(data['arr_0']):
    img = Image.fromarray( arr )
    img = img.resize( image_display_size )
    img.save( f"visualize_samples/test_{i}.jpg" )

# generate image grid
im_tensor_all = torch.from_numpy(data['arr_0']).float()
im_tensor_all = torch.transpose(im_tensor_all, 1,3)
im_tensor_all = torch.transpose(im_tensor_all, 2,3)
cv2.imwrite(visualization_dir+"/fake_images.png", np.flip(torchvision.
    ↪ utils.make_grid(im_tensor_all, nrow=4, padding=1, pad_value=1,
    ↪ normalize=True).cpu().numpy().transpose(1,2,0),2)*255)
```

Sample real images, sample fake images by GAN, and sample fake images by Diffusion (with 1000 diffusion time steps) can be found in Fig. 3, Fig. 4, and Fig. 5 respectively. Comparing the images from Fig. 4 and Fig. 5, the GAN produces more blurred parts, especially around the hair areas; the outputs of both GAN and Diffusion models are generally good, with the basic qualities of the human face; both models tend to output solid color backgrounds; both models output some noise with weird-value pixels.

Figure 3: Sample real images.

Figure 4: Sample fake images by GAN.

Figure 5: Sample fake images by Diffusion.

1.3.2 Quantitative evaluation

In this experiment, a quantitative evaluation using Frechet Inception Distance (FID) is performed. The code is shown below. 2048 fake images are generated in `generate_fake` function. Then the FID is calculated in `calculate_fid` function using `calculate_frechet_distance`.

```
...
# generate fake images
def generate_fake(args, device):
    # create a folder to save images
    dir_name_for_results = args.fake_path
    if os.path.exists(dir_name_for_results):
```

```
        files = glob.glob(dir_name_for_results + "/*")
        for file in files:
            if os.path.isfile(file):
                os.remove(file)
            else:
                files = glob.glob(file + "/*")
                list(map(lambda x: os.remove(x), files))
    else:
        os.mkdir(dir_name_for_results)
    # load the trained network
    generator = torch.load('{}generator.pth'.format(args.result_path)).
        ↪ to(device).eval()
    # create NumPy array to save image data
    images = np.zeros((args.n_fake_sample, args.img_size[0], args.
        ↪ img_size[0], 3), dtype=np.uint8)
    # generate images one by one
    with torch.no_grad():
        for i in range(args.n_fake_sample):
            noise = torch.randn(1, args.nz, 1, 1, device=device)
            fake = generator(noise).detach().cpu()
            images[i,:,:,:] = (np.flip(fake[0,:,:,:].detach().cpu().
                ↪ numpy().transpose(1,2,0),2).clip(-1,1)+1)*127.5
            # save current image
            cv2.imwrite('{}fake_imgs_{}.jpg'.format(args.fake_path,i),
                ↪ images[i,:,:,:])
    # save NumPy array
    np.save('{}imgs_fake_dcgan.npy'.format(args.result_path), images)

# calculate FID
def calculate_fid(args, device):
    # retrive image paths
    real_path = sorted(glob.glob(os.path.join(args.imgs_path, '1/*.jpg'
        ↪ )))[:args.n_fake_sample] # real images
    # fake_path = sorted(glob.glob(os.path.join(args.fake_path, '*.jpg
        ↪ '))) # fake images by DCGAN
    fake_path = sorted(glob.glob('../ExamplesDiffusion/
        ↪ visualize_samples/*.jpg')) # fake images by Diffusion
    # calculate activation statistics
    dims = 2048
    block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
    model = InceptionV3([block_idx]).to(device)
    m1, s1 = calculate_activation_statistics(real_path, model, device=
        ↪ device)
    m2, s2 = calculate_activation_statistics(fake_path, model, device=
        ↪ device)
    # calculate Frechet Inception Distance
    fid_value = calculate_frechet_distance(m1, s1, m2, s2)
    print(f'FID: {fid_value:.2f}')
...
```

The FID for GAN and Diffusion models can be found in Tab. 1. The Diffusion model has a lower FID.

Table 1: FID for GAN and Diffusion models.

Model	FID
GAN	80.90
Diffusion	49.49

1.3.3 Discussion

According to the comparison, the Diffusion model is better in terms of image quality. First, the outputs from the Diffusion model have clearer and sharper boundaries, and they contain more details of the faces. Second, the FID of the diffusion model is lower.

However, the Diffusion is much slower. It takes several hours for Diffusion to generate 2048 fake images, while it only needs less than 10 seconds for GAN to generate the same amount of fake images.

To summarize, the trade-off between the image quality and running time needs to be considered. If the image quality is more important, the Diffusion model would be a better choice, but if the running time is more important, the GAN model could be a better choice.

2 Source Code

```
import os
import numpy as np
import random
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torchvision
import torchvision.transforms as tvt
import torchvision.transforms.functional as tvtF
import cv2
import imageio
import matplotlib.pyplot as plt
import glob
from pytorch_fid.fid_score import calculate_activation_statistics,
    ↪ calculate_frechet_distance
from pytorch_fid.inception import InceptionV3
import argparse

# set ramdom seed
def set_random_seed(seed):
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmarks = False
    os.environ['PYTHONHASHSEED'] = str(seed)

# create data loader
```

```
def create_dataloader(args):
    dataset = torchvision.datasets.ImageFolder(root=args.imgs_path,
                    transform = tvt.Compose([tvt.Resize(args.img_size),
                                             tvt.ToTensor(),
                                             tvt.Normalize((0.5, 0.5,
                                                ↪ 0.5), (0.5, 0.5, 0.5)
                                                ↪ )
                                             ]))
    train_dataloader = DataLoader(dataset, batch_size=args.batch_size,
       ↪ shuffle=True, num_workers=args.n_workers)
    return train_dataloader

# class for generator
class GeneratorDCGAN(nn.Module):
    def __init__(self):
        super().__init__()
        # transposed convolutional layer with batch normalizaiton and
           ↪ ReLU activation
        def trans_conv_layer(in_channels, out_channels, kernel_size,
           ↪ stride=1, padding=0):
            return nn.Sequential(nn.ConvTranspose2d(in_channels,
               ↪ out_channels, kernel_size=kernel_size, stride=stride,
               ↪  padding=padding, bias=False),
                                 nn.BatchNorm2d(out_channels),
                                 nn.ReLU(inplace=True))
        # define layers
        self.latent2image = trans_conv_layer(100, 512, 4, 1, 0)
        self.upsampler2 = trans_conv_layer(512, 256, 4, 2, 1)
        self.upsampler3 = trans_conv_layer(256, 128, 4, 2, 1)
        self.upsampler4 = trans_conv_layer(128, 64, 4, 2, 1)
        self.upsampler5 = nn.ConvTranspose2d(64, 3, kernel_size=4,
           ↪ stride=2, padding=1, bias=False)
        self.tanh = nn.Tanh()
    def forward(self, z):
        z = self.latent2image(z)
        z = self.upsampler2(z)
        z = self.upsampler3(z)
        z = self.upsampler4(z)
        z = self.upsampler5(z)
        z = self.tanh(z)
        return z

# class for discriminator
class DiscriminatorDCGAN(nn.Module):
    def __init__(self):
        super().__init__()
        # convolutional layer with batch normalizaiton (optional) and
           ↪ ReLU activation
        def conv_layer(in_channels, out_channels, kernel_size, stride
           ↪ =1, padding=0, bn=True):
            if bn:
                return nn.Sequential(nn.Conv2d(in_channels,
                   ↪ out_channels, kernel_size=kernel_size, stride=
                   ↪ stride, padding=padding),
```

```
                                        nn.BatchNorm2d(out_channels),
                                        nn.LeakyReLU(negative_slope=0.2,
                                            ↪ inplace=True))
            else:
                return nn.Sequential(nn.Conv2d(in_channels,
                    ↪ out_channels, kernel_size=kernel_size, stride=
                    ↪ stride, padding=padding),
                                        nn.LeakyReLU(negative_slope=0.2,
                                            ↪ inplace=True))
        # define layers
        self.conv1 = conv_layer(3, 64, 4, 2, 1, False) # first
            ↪ convolutional layer does not have batch normalization
        self.conv2 = conv_layer(64, 128, 4, 2, 1)
        self.conv3 = conv_layer(128, 256, 4, 2, 1)
        self.conv4 = conv_layer(256, 512, 4, 2, 1)
        self.conv5 = nn.Conv2d(512, 1, 4, 1, 0)
        self.sig = nn.Sigmoid()
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.sig(x)
        return x

# initialize weights
def weight_init(m):
    classname = m.__class__.__name__
    # initialize (transposed) convolutional layers
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    # initialize batch normalization layers
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)

# plot the learning curves
def plot_learning_curve(args, G_loss_list, D_loss_list):
    # generate x-axis
    x = [i for i in range(len(G_loss_list))]
    plt.figure(figsize=(10, 5))
    # plot each curve
    plt.plot(x, G_loss_list, label='Generator')
    plt.plot(x, D_loss_list, label='Discriminator')
    plt.xlabel('iterations')
    plt.ylabel('losses')
    plt.legend()
    cf = plt.gcf()
    cf.savefig('{}loss.png'.format(args.result_path), bbox_inches='
        ↪ tight', format='png')

# train the model
def train(args, device):
```

```
    # create a folder to save results
    dir_name_for_results = args.result_path
    if os.path.exists(dir_name_for_results):
        files = glob.glob(dir_name_for_results + "/*")
        for file in files:
            if os.path.isfile(file):
                os.remove(file)
            else:
                files = glob.glob(file + "/*")
                list(map(lambda x: os.remove(x), files))
    else:
        os.mkdir(dir_name_for_results)
    # create training data loader
    train_dataloader = create_dataloader(args)
    # fixed noise to track performance during training
    noise_fixed = torch.randn(args.batch_size, args.nz, 1, 1, device=
        ↪ device)
    # create model instance
    generator = GeneratorDCGAN().to(device)
    discriminator = DiscriminatorDCGAN().to(device)
    # iniitalize weights
    generator.apply(weight_init)
    discriminator.apply(weight_init)
    # count number of learnable parameters
    num_learnable_params_gen = sum(p.numel() for p in generator.
        ↪ parameters() if p.requires_grad)
    print('The number of learnable parameters in the Generator: %d'%
        ↪ num_learnable_params_gen)
    num_learnable_params_disc = sum(p.numel() for p in discriminator.
        ↪ parameters() if p.requires_grad)
    print('The number of learnable parameters in the Discriminator: %d'
        ↪ %num_learnable_params_disc)
    # count number of layers
    num_layers_gen = len(list(generator.parameters()))
    print('The number of layers in the generator: %d'%num_layers_gen)
    num_layers_disc = len(list(discriminator.parameters()))
    print('The number of layers in the discriminator: %d'%
        ↪ num_layers_disc)
    # use Adam optimizer for both networks
    optimizer_G = torch.optim.Adam(generator.parameters(), lr=args.lr,
        ↪ betas=(args.beta1,0.99))
    optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=args.
        ↪ lr, betas=(args.beta1,0.99))
    # use BCE loss
    criterion = nn.BCELoss()
    # list to save losses
    G_loss_list = []
    D_loss_list = []
    # labels for real and fake images
    real_label_val = 1
    fake_label_val = 0
    # track iteration index
    iters = 0
    # list to save generated fake images during training
```

```
    imgs_list = []
    for epoch in range(args.n_epochs):
        # list to save losses for printing
        G_loss_list_temp = []
        D_loss_list_temp = []
        for i, data in enumerate(train_dataloader):
            imgs_real = data[0].to(device)
            # actual number of images in current batch
            b_size = imgs_real.size(0)
            # create labels for real and fake images
            label_real = torch.full((b_size,), real_label_val, dtype=
                ↪ torch.float, device=device)
            label_fake = torch.full((b_size,), fake_label_val, dtype=
                ↪ torch.float, device=device)
            # train discriminator
            discriminator.zero_grad()
            output = discriminator(imgs_real).view(-1)
            D_loss_real = criterion(output, label_real)
            D_loss_real.backward() # gradient from real images
            noise = torch.randn(b_size, args.nz, 1, 1, device=device)
            fakes = generator(noise) # generator fake images
            output = discriminator(fakes.detach()).view(-1)
            D_loss_fake = criterion(output, label_fake)
            D_loss_fake.backward() # gradient from fake images
            optimizer_D.step() # update discriminator
            D_loss = D_loss_real + D_loss_fake
            D_loss_list_temp.append(D_loss)
            # train generator
            generator.zero_grad()
            output = discriminator(fakes).view(-1)
            G_loss = criterion(output, label_real)
            G_loss.backward() # gradient from generated images
            optimizer_G.step() # update generator
            G_loss_list_temp.append(G_loss)
            # periodically print losses
            if i%args.print_every == args.print_every-1:
                mean_G_loss = torch.mean(torch.FloatTensor(
                    ↪ G_loss_list_temp))
                mean_D_loss = torch.mean(torch.FloatTensor(
                    ↪ D_loss_list_temp))
                print('[epoch=%d/%d iter=%4d] mean_D_loss=%7.4f
                    ↪ mean_G_loss=%7.4f]'%((epoch+1),args.n_epochs,(i
                    ↪ +1),mean_D_loss,mean_G_loss))
                G_loss_list_temp = []
                D_loss_list_temp = []
            # save losses
            G_loss_list.append(G_loss.item())
            D_loss_list.append(D_loss.item())
            # periodically visualize generated images
            if (iters%args.generate_every == 0) or ((epoch == args.
                ↪ n_epochs-1) and (i == len(train_dataloader)-1)):
                with torch.no_grad():
                    fake = generator(noise_fixed).detach().cpu()
                imgs_list.append(torchvision.utils.make_grid(fake,
```

```
                    ↪ padding=1, pad_value=1, normalize=True))
212             iters += 1
213     # save the trained network
214     torch.save(generator, '{}generator.pth'.format(args.result_path))
215     torch.save(discriminator, '{}discriminator.pth'.format(args.
            ↪ result_path))
216     # plot learning curve
217     plot_learning_curve(args, G_loss_list, D_loss_list)
218     # generate GIF
219     imgs = []
220     for imgobj in imgs_list:
221         img = tvtF.to_pil_image(imgobj)
222         imgs.append(img)
223     imageio.mimsave('{}generation_animation.gif'.format(args.
            ↪ result_path), imgs, fps=5)
224     # visualize real image samples from the dataset
225     real_batch = next(iter(train_dataloader))[0]
226     cv2.imwrite('{}real_imgs.png'.format(args.result_path), np.flip(
            ↪ torchvision.utils.make_grid(real_batch.to(device), padding=1,
            ↪  pad_value=1, normalize=True).cpu().numpy().transpose(1,2,0)
            ↪ ,2)*255)
227     # visualize fake image samples
228     cv2.imwrite('{}fake_imgs.png'.format(args.result_path), np.flip(
            ↪ imgs_list[-1].numpy().transpose(1,2,0),2).clip(0,1)*255)
229
230 # generate samples
231 def generate_img_demo(args, device):
232     # change the batch size to 16 for the 4 by 4 demo
233     args.batch_size = 16
234     # create training data loader
235     train_dataloader = create_dataloader(args)
236     # load the trained network
237     generator = torch.load('{}generator.pth'.format(args.result_path)).
            ↪ to(device).eval()
238     # visualize real image samples from the dataset
239     real_batch = next(iter(train_dataloader))[0]
240     cv2.imwrite('{}real_imgs_demo.png'.format(args.result_path), np.
            ↪ flip(torchvision.utils.make_grid(real_batch.to(device), nrow
            ↪ =4, padding=1, pad_value=1, normalize=True).detach().cpu().
            ↪ numpy().transpose(1,2,0),2)*255)
241     # visualize fake image samples
242     noise = torch.randn(args.batch_size, args.nz, 1, 1, device=device)
243     with torch.no_grad():
244         fake = generator(noise).detach().cpu()
245     cv2.imwrite('{}fake_imgs_demo.png'.format(args.result_path), np.
            ↪ flip(torchvision.utils.make_grid(fake, nrow=4, padding=1,
            ↪ pad_value=1, normalize=True).numpy().transpose(1,2,0),2).clip
            ↪ (0,1)*255)
246
247 # generate fake images
248 def generate_fake(args, device):
249     # create a folder to save images
250     dir_name_for_results = args.fake_path
251     if os.path.exists(dir_name_for_results):
```

```
252             files = glob.glob(dir_name_for_results + "/*")
253             for file in files:
254                 if os.path.isfile(file):
255                     os.remove(file)
256                 else:
257                     files = glob.glob(file + "/*")
258                     list(map(lambda x: os.remove(x), files))
259         else:
260             os.mkdir(dir_name_for_results)
261         # load the trained network
262         generator = torch.load('{}generator.pth'.format(args.result_path)).
                ↪ to(device).eval()
263         # create NumPy array to save image data
264         images = np.zeros((args.n_fake_sample, args.img_size[0], args.
                ↪ img_size[0], 3), dtype=np.uint8)
265         # generate images one by one
266         with torch.no_grad():
267             for i in range(args.n_fake_sample):
268                 noise = torch.randn(1, args.nz, 1, 1, device=device)
269                 fake = generator(noise).detach().cpu()
270                 images[i,:,:,:] = (np.flip(fake[0,:,:,:].detach().cpu().
                        ↪ numpy().transpose(1,2,0),2).clip(-1,1)+1)*127.5
271                 # save current image
272                 cv2.imwrite('{}fake_imgs_{}.jpg'.format(args.fake_path,i),
                        ↪ images[i,:,:,:])
273         # save NumPy array
274         np.save('{}imgs_fake_dcgan.npy'.format(args.result_path), images)
275 
276 # calculate FID
277 def calculate_fid(args, device):
278     # retrive image paths
279     real_path = sorted(glob.glob(os.path.join(args.imgs_path, '1/*.jpg'
            ↪ )))[:args.n_fake_sample] # real images
280     fake_path = sorted(glob.glob(os.path.join(args.fake_path, '*.jpg'))
            ↪ ) # fake images by DCGAN
281     # fake_path = sorted(glob.glob('../ExamplesDiffusion/
            ↪ visualize_samples/*.jpg')) # fake images by Diffusion
282     # calculate activation statistics
283     dims = 2048
284     block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
285     model = InceptionV3([block_idx]).to(device)
286     m1, s1 = calculate_activation_statistics(real_path, model, device=
            ↪ device)
287     m2, s2 = calculate_activation_statistics(fake_path, model, device=
            ↪ device)
288     # calculate Frechet Inception Distance
289     fid_value = calculate_frechet_distance(m1, s1, m2, s2)
290     print(f'FID: {fid_value:.2f}')
291 
292 if __name__ == '__main__':
293     parser = argparse.ArgumentParser()
294     # '1' -- train the DCGAN model
295     # '2' -- generate 4 by 4 demos
296     # '3' -- generate 2048 fake images
```

```
    # '4' -- calculate FID
    parser.add_argument('--task', type=str, default='1',\
        help='choose a task', choices=['1','2','3','4'])
    parser.add_argument('--imgs_path', type=str, default='./celeba/',
        ↪ help='dataset path')
    parser.add_argument('--result_path', type=str, default='./
        ↪ celeba_result/', help='dataset path')
    parser.add_argument('--fake_path', type=str, default='./celeba_fake
        ↪ /', help='fake path')
    parser.add_argument('--print_every', type=int, default=100, help='
        ↪ printing frequency')
    parser.add_argument('--generate_every', type=int, default=500, help
        ↪ ='generating frequency')
    parser.add_argument('--n_epochs', type=int, default=30, help='batch
        ↪  size')
    parser.add_argument('--img_size', type=tuple, default=(64,64), help
        ↪ ='image size')
    parser.add_argument('--batch_size', type=int, default=32, help='
        ↪ batch size')
    parser.add_argument('--n_workers', type=int, default=2, help='
        ↪ number of workers')
    parser.add_argument('--nz', type=int, default=100, help='number of
        ↪ channels of 1x1 input noise vectors for GeneratorDCGAN')
    parser.add_argument('--lr', type=float, default=1e-4, help='
        ↪ learning rate')
    parser.add_argument('--beta1', type=float, default=0.5, help='
        ↪ beta_1 for Adam optimizer')
    parser.add_argument('--n_fake_sample', type=int, default=2048, help
        ↪ ='number of fake images')
    parser.add_argument('--cuda', type=str, default='cpu', help='Enable
        ↪  cuda')
    args = parser.parse_args()

    device = torch.device(args.cuda)
    seed = 0
    set_random_seed(seed)

    if args.task == '1':
        train(args, device)
    if args.task == '2':
        generate_img_demo(args, device)
    if args.task == '3':
        generate_fake(args, device)
    if args.task == '4':
        calculate_fid(args, device)
```